Word Level Symbolic Model Checking

A New approach for Verifying Arithmetic Circuits

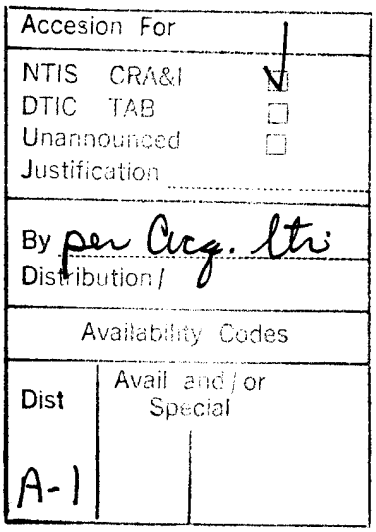
E. Clarke X. Zhao

May, 1995
CMU-CS-95-161

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by the National Science Foundation under Grant No. CCR-9217549, by the Semiconductor Research Corporation under Contract No. 94-DJ-294, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under Grant No. F33615-93-1-1330. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory or the United States Government.



Keywords: automatic verification, temporal logic, model checking, binary decision diagrams, multi-terminal binary decision diagrams, binary moment diagrams, hybrid decision diagrams, word level properties, arithmetic circuit, Pentium, division circuit

Abstract

The highly-publicized division error in the Pentium has emphasized the importance of formal verification of arithmetic operations. Symbolic model checking techniques based on binary decision diagrams (BDDs) have been successful in verifying control logic. However, lack of proper representation for functions that map boolean vectors into integers has prevented this technique from being used for verifying arithmetic circuits.

We have used **hybrid decision diagrams** to represent the integer functions that occur in the arithmetic circuit verification. For the state variables corresponding to data bits, our representation behaves like a binary moment diagram (BMD) while for the state variables corresponding to control signals, it behaves like a multi-terminal BDD (MTBDD). By using this representation, we are able to handle circuits with both control logic and wide data paths.

We have extended the symbolic model checking system SMV so that it can also handle properties involving relationships among data words. In the original SMV system, atomic formulas could only contain state variables. In the extended system, we allow atomic formulas to be equations or inequalities between expressions as well. These expressions are represented as hybrid decision diagrams.

The extended model checking system enables us to verify circuits for division and square root computation that are based on the SRT algorithm used by the Pentium. We are able to handle both the control logic and the data paths. The total number of state variables exceeds 600 (which is much larger than any circuit previously checked by SMV).

1. Introduction

Proving the correctness of arithmetic operations has always been an important problem. The importance of this problem has been recently emphasized by the highly-publicized division error in the Pentium. In order to verify such circuits, it is necessary to represent and manipulate functions that map boolean vectors to integer values. In this paper, we describe how to represent and manipulate such functions efficiently using *Multi-Terminal Binary Decision Diagrams* (MTBDDs) [8]. An MTBDD is like an ordinary Binary Decision Diagram (BDD)[3] except that the terminal nodes can be arbitrary integer values instead of just 0 and 1. We have also investigated a technique for representing such integer valued functions by BDD arrays. Unfortunately, both representations have problems when they are used for verifying arithmetic circuits. For the functions that arise in this type of application, the number of possible values is exponential in the number of bits. Therefore, the MTBDDs also have exponential size. Since the BDD size for the middle bit of a combinational multiplier is exponential in the length of its operands, the BDD array representation is exponential for multiplication. Moreover, arithmetic operations on BDD arrays are very expensive.

Bryant and Chen have developed another representation called the *Binary Moment Diagram* (BMD)[4]. They use the expansion $f = f|_{x=0} + xf'$, where f' is equal to $f|_{x=1} - f|_{x=0}$, instead of the Shannon expansion. This gives a compact representation for certain functions that have exponential size MTBDDs. They have used this *word level* representation to verify the data paths of some arithmetic circuits. The BMD representation for both the circuit and the specification are constructed and compared. The circuit is correct if both BMDs are exactly the same. However, depending on the implementation and the control logic, there can be cases in which the circuit is correct but the BMDs are not identical. Another problem is that this approach cannot handle inequalities. Thus, it is impossible to check some of the properties that are needed in order to avoid the Pentium error.

We first show that the BMD of a function is the MTBDD that results from applying the inverse Reed-Muller transformation [12] to the function. The transformation can be computed using the techniques that we have previously developed for manipulating large matrices [8]. The transformation matrix in this case is the Kronecker product [2] of a number of identical 2×2 matrices. We show that the Kronecker products of other 2×2 matrices behave in a similar way. In fact, the transformations obtained from Kronecker products of other matrices will in many cases be more concise than the BMD. We have further generalized this idea so that the transformation matrix can be the Kronecker product of different matrices. In this way, we obtain a representation, called the Hybrid Decision Diagram (HDD), that is more concise than either the MTBDD or the BMD. In addition to algorithms for performing arithmetic operations, we have developed an efficient algorithm to compute the set of variable assignments that satisfy an arithmetic relation. For the class of linear functions, which includes many of the functions that occur in practice, such operations are guaranteed to have complexity that is polynomial in the width of the data words.

Our representation for functions that map boolean vectors into the integers enables us to extend *temporal logic model checking* [6, 7] so that it can handle arithmetic circuits. In traditional model checking systems, specifications are expressed in a propositional temporal logic, and circuit designs and protocols are modeled as state-transition systems. An efficient

search procedure is used to determine automatically if the specifications are satisfied by the transition systems. The main disadvantage of this approach is the state explosion which can occur if the system being verified has many components that can make transitions in parallel. Recently, the size of the transition systems that can be verified by model checking techniques has increased dramatically because of the use of BDDs [5]. Although such *symbolic model checking techniques* have been successful in verifying control logic, these techniques cannot be directly used for verifying arithmetic circuits.

One of the main reasons that the symbolic model checking systems cannot handle arithmetic circuits is the lack of a concise representation for expressions that involve words with integer values. We have used hybrid decision diagrams to represent the integer functions that occur in the arithmetic circuit verification. For the state variables corresponding to data bits, our representation behaves like a BMD while for the state variables corresponding to control signals, it behaves like an MTBDD. By using this representation, we are able to handle circuits with both control logic and wide data paths. We have extended the symbolic model checking system SMV [11] so that it can also handle properties involving relationships among data words. In the original SMV system, atomic formulas could only contain state variables. In the extended system, we allow atomic formulas to be equations or inequalities between expressions as well. These expressions are represented as hybrid decision diagrams.

In the word level model checking system, propositions denoting nodes in circuits are represented as BDDs and are computed in exactly the same way as in the original symbolic model checking system. Words are arrays of propositions, each of which corresponds to a single bit. Expressions are composed of arithmetic operations applied to words. Hybrid decision diagrams can be computed for words and expressions using the algorithms for arithmetic operations. Atomic formulas can be relations between expressions, and their BDD representations can be computed by the algorithm that handles arithmetic relations. After the BDD representations for the atomic formulas are generated, the BDDs for static formulas and temporal formulas are computed in the same way as in ordinary model checking. In particular, the fixpoint computations are exactly the same in both cases.

By using the word level model checking system, we have successfully verified circuits for division and square root computation that are based on the SRT algorithm used by the Pentium. We are able to handle both the control logic and the data paths. All of the states in the finite state machine for the control logic have been verified. Moreover, we have proved invariant properties that guarantee the correctness of the data values and prevent overflows. The total number of state variables exceeds 600 (which is much larger than any circuit previously checked by SMV).

This paper is organized as follows: In Section 2, we discuss different techniques for representing functions that map boolean vectors into the integers. In Section 3, we give the logic that is used for specifying the properties involving the data values. In this section, we also describe how formulas can be represented by a special class of hybrid decision diagrams. In Sections 4 and 5 algorithms for handling arithmetic operations and arithmetic relations are given. Section 4 gives the algorithms for computing addition, multiplication and the `if-then-else` operation for this representation. Section 5 gives an algorithm that computes the BDD representation for the set of variable assignments that satisfy an equation or an inequality. In Section 6, we discuss how the word level model checking is performed. We

illustrate the power of our technique in Section 7 by showing how word level model checking can be used to verify a division circuit based on the radix-4 SRT algorithm that is similar to the one used by the Pentium. The paper concludes in Section 8 with a discussion of possible future research directions.

2. Hybrid Decision Diagrams

Ordered binary decision diagrams (BDDs) are a canonical representation for boolean formulas proposed by Bryant [3]. They are often substantially more compact than traditional normal forms such as conjunctive normal form and disjunctive normal form. They can also be manipulated very efficiently. Hence, BDDs have become widely used for a variety of CAD applications, including symbolic simulation, verification of combinational logic and, more recently, verification of sequential circuits.

A BDD is similar to a binary decision tree, except that its structure is a directed acyclic graph rather than a tree, and there is a strict total order placed on the occurrence of variables as one traverses the graph from root to leaf. Algorithms of linear complexity exist for computing BDD representations of $\neg f$ and $f \vee g$ from the BDDs for the formulas f and g.

Let $f : B^m \rightarrow Z$ be a function that maps boolean vectors of length m into integers. Suppose $n_1, \ldots, n_N$ are the possible values of f. The function f partitions the space B^m of boolean vectors into N sets $\{S_1, \cdots, S_N\}$, such that $S_i = \{\, \bar{x} \mid f(\bar{x}) = n_i \,\}$. Let f_i be the characteristic function of S_i, we say that f is in *normal form* if $f(\bar{x})$ is represented as $\sum_{i=1}^{N} f_i(\bar{x}) \cdot n_i$. This sum can be represented as a BDD with integers as its terminal nodes. We call such DAGs *Multi-Terminal BDDs* (MTBDDs) [8, 1].

Let $f : B^m \rightarrow Z$ be a function that maps boolean vectors of length m into integers. Suppose $n_1, \ldots, n_N$ are the possible values of f. The function f partitions the space B^m of boolean vectors into N sets $\{S_1, \cdots, S_N\}$, such that $S_i = \{\, \bar{x} \mid f(\bar{x}) = n_i \,\}$. Let f_i be the characteristic function of S_i, we say that f is in *normal form* if $f(\bar{x})$ is represented as $\sum_{i=1}^{N} f_i(\bar{x}) \cdot n_i$. This sum can be represented as a BDD with integers as its terminal nodes. We call such DAGs *Multi-Terminal BDDs* (MTBDDs) [8, 1].

Any arithmetic operation $\odot$ on MTBDDs can be performed in the following way. There is an efficient algorithm that computes the operation in time linear to the sizes of the MTBDDs of both operands.

$$
\begin{aligned}
h(\bar{x}) &= f(\bar{x}) \odot g(\bar{x}) \\
&= \sum_{i=1}^{N} f_i(\bar{x}) \cdot n_i \odot \sum_{j=1}^{N'} g_j(\bar{x}) \cdot n'_j \\
&= \sum_{i=1}^{N} \sum_{j=1}^{N'} f_i(\bar{x}) g_j(\bar{x}) (n_i \odot n'_j) \\
&= \sum_{k=1}^{N''} \bigvee_{n_i \odot n'_j = n''_k} f_i(\bar{x}) g_j(\bar{x}) n''_k
\end{aligned}
$$

Functions that map boolean vectors into the integers can also be represented as arrays

of BDDs. These BDDs have boolean values and each corresponds to one bit of the binary representation of the function value. In general, it is quite expensive to perform operations using this representation.

Let M be a $2^k \times 2^l$ matrix over Z. It is easy to see that M can be represented as a function $M : B^{k+l} \rightarrow Z$, such that $M_{ij} = M(\bar{x}, \bar{y})$, where $\bar{x}$ is the bit vector for i and $\bar{y}$ is the bit vector for j. Therefore, matrices with integer values can be represented as integer valued functions using the representation shown above. We can also perform various matrix operations using our MTBDD representation. In particular, matrix multiplication can be computed in the following way: Suppose that two matrices A and B have dimensions $2^k \times 2^l$ and $2^l \times 2^m$, respectively. Let $C = A \times B$ be the product of A and B, then C will have dimension $2^k \times 2^m$. If we treat A and B as integer-valued functions, we can compute the product matrix C as

$$C(\bar{x}, \bar{z}) = \sum_{\bar{y}} A(\bar{x}, \bar{y}) B(\bar{y}, \bar{z}),$$

where $\sum_{\bar{y}}$ means "sum over all possible assignments to $\bar{y}$". Although this operation works well in many cases, the worst case complexity can be exponential in the number of variables.

Recently, Bryant and Chen[4] have developed a new representation for functions that map boolean vectors to integer values. This representation is called the Binary Moment Diagram (BMD) of the function. Instead of the Shannon expansion $f = xf_1 + (1 - x)f_0$, they use the expansion $f = f_0 + xf'$, where f' is equal to $f_1 - f_0$. After merging the common subexpressions, a DAG representation for the function is obtained. They prove in their paper that this gives a compact representation for certain functions which have exponential size if represented by MTBDDs directly.

There is a close relationship between this representation and the inverse Reed-Muller transformation [12]. The matrix for the inverse Reed-Muller transformation is defined recursively by

$$S_0 = 1 \qquad S_n = \begin{pmatrix} S_{n-1} & 0 \\ -S_{n-1} & S_{n-1} \end{pmatrix}$$

which has a linear MTBDD representation. Let $\bar{i} \in B^n$ be the binary representation of integer $0 \leq i < 2^n$. A function $f : B^n \rightarrow N$ can be represented as a column vector where the value of the ith entry is $f(\bar{i})$. We will not distinguish between a function and its corresponding column vector. The inverse Reed-Muller transformation can be obtained by multiplying the transformation matrix and the column vector $\widehat{f} = S \times f$ using the technique described in previous section.

Theorem 1 *The MTBDD of $\widehat{f}$ is isomorphic to the BMD of f.*

Proof: The theorem is easy to prove by induction on the number of variables.

Base Case: If the number of variables is 0, the function is a constant and $\widehat{f} = f$. Both the MTBDD of $\widehat{f}$ and the BMD for f are terminal nodes and therefore isomorphic.

Induction Step: Let $f : B^n \rightarrow N$. The roots of both the BMD for f and the MTBDD for $\widehat{f}$ are x_n. The left child of the root of the BMD for f is the BMD for $f|_{x_n=0}$, while the right child is the BMD for $f|_{x_n=1} - f|_{x_n=0}$. When f is represented as a column vector,

the upper half is $f|_{x_n=0}$ and the bottom half is $f|_{x_n=1}$. The inverse Reed-Muller matrix is $\begin{pmatrix} S_{n-1} & 0 \\ -S_{n-1} & S_{n-1} \end{pmatrix}$. The result of the transformation is therefore:

$$\begin{pmatrix} S_{n-1} & 0 \\ -S_{n-1} & S_{n-1} \end{pmatrix} \times \begin{pmatrix} f|_{x_n=0} \\ f|_{x_n=1} \end{pmatrix} = \begin{pmatrix} S_{n-1} \times f|_{x_n=0} \\ S_{n-1} \times (f|_{x_n=1} - f|_{x_n=0}) \end{pmatrix}$$

If this vector is represented by MTBDD, the left child is the MTBDD for the inverse Reed-Muller transform of $f|_{x_n=0}$ and the right child is the MTBDD for the inverse Reed-Muller transform of $f|_{x_n=1} - f|_{x_n=0}$. By induction hypothesis, both children are isomorphic to the children of the root of the BMD for f. Therefore the BMD of f is isomorphic to the MTBDD for $\hat{f}$. □

The inverse Reed-Muller matrix can be represented as the Kronecker product [2] of n identical 2×2 matrices:

$$S_n = \begin{pmatrix} S_{n-1} & 0 \\ -S_{n-1} & S_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \otimes S_{n-1} = \underbrace{\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \otimes \cdots \otimes \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}}_{n}$$

The inverse Reed-Muller transformation is not unique in this respect. Other transformations that are defined as Kronecker products of 2×2 matrices may also provide concise representations for functions mapping boolean vectors into integers. In fact, the Kronecker product of any non-singular 2×2 matrices can be used as a transformation matrix and will produce a canonical representation for the function. Moreover, if the transformation matrix is a Kronecker product of different 2×2 matrices, we still have a canonical representation of the function. We call transformations obtained from such matrices *hybrid transformations.*

A similar strategy has been tried by Becker [10]. However, his technique only works for the boolean domain. When using his technique, all of the transformation matrices, the original function and the resulting function must have boolean values. Our technique, on the other hand, works over the integers. By allowing integer values, we can handle a wider range of functions. Moreover, we can obtain larger reduction factors since we have more choices for transformation matrices.

We can apply this idea to reduce the size of the BDD representation of the functions. Since there is no known polynomial algorithm to find the hybrid Kronecker transformation that minimizes BDD size, we use a greedy algorithm to reduce the size. If we restrict the entries in the matrix to the set $\{0, 1, -1\}$, then there are six matrices we can try. For each variable, we select the matrix that gives the smallest BDD size. The BDDs obtained from such transformations are called Hybrid Decision Diagrams (HDDs). We have tried this method on the ISCAS85 benchmark circuits. In some cases we have been able to reduce the size of the BDD representation by a factor of 1300. However, reductions of this magnitute usually occur when the original function has a bad variable ordering. If dynamic variable ordering is used, then our method gives a much smaller reduction factor.

3. The Logic

Symbolic model checking techniques based on Binary Decision Diagrams (BDDs) have been successful in verifying control logic [5]. However, lack of proper representation for functions that map boolean vectors into integers has prevented this technique from being used for verifying arithmetic circuits. We have experimented with the different representations that are introduced in previous sections. Unfortunately, there are fundamental problems with applying either the MTBDD or the BDD array representations for verification of arithmetic circuits. For the functions that arise in this type of application, the number of possible values is exponential in the number of bits. Therefore, the MTBDDs also have exponential size. On the other hand, arithmetic operations on BDD arrays are very expensive. In particular, since the BDD size for the middle bit of a combinational multiplier is exponential in the length of its operands, the BDD array representation is exponential for multiplication.

Bryant and Chen [4] have shown that the BMD gives a compact representation for certain functions that have exponential size MTBDDs. They have used this representation to verify the data paths of some arithmetic circuits. They are able to conclude that a circuit is correct if the BMDs for the circuit and the specification are exactly the same. However, depending on the implementation and the control logic, there can be cases in which the circuits are correct but the BMDs are not identical. Moreover, since their technique cannot handle inequalities, it is impossible to check some of the properties that are needed in order to avoid the Pentium error.

We have used hybrid decision diagrams to represent the integer functions that occur in the arithmetic circuit verification. In particular, for the state variables corresponding to data bits, we use the inverse Reed-Muller transform while for the state variables corresponding to control signals, we use the identity transform. Therefore, for data variables, this representation behaves like a BMD while for control variables, it behaves like a MTBDD. By using this representation, we are able to handle circuits with both control logic and wide data paths. Since this representation is a special case of the hybrid decision diagrams, all the algorithms mentioned in previous sections can be applied.

By using this representation, we have extended the symbolic model checking system SMV [11] so that it can also handle properties involving relationships among data words. In the original SMV system, atomic formulas can only contain state variables. In the extended system, we allow atomic formulas be equations or inequalities between expressions as well. These expressions are represented as hybrid BDDs. The logic that we use is the follows:

- Atomic propositions: $Ap = \{p_1, \ldots, p_k\}$
- Propositional formulas: $Prop ::= Ap \mid Prop \wedge Prop \mid \neg Prop$
- Words: $Word ::= (Prop, Prop, \ldots, Prop)$
- Expressions:
  $Exp ::= Constant \mid Word \mid \text{next}(Word) \mid Exp \odot Exp \mid \text{if } SF \text{ then } Exp \text{ else } Exp$,
  where $\odot$ can be $+$, $-$, or $\times$.
- Atomic Formulas: $AF ::= Ap \mid \{\mathbf{A} \mid \mathbf{E}\}(Exp \sim Exp)$, where $\sim$ can be $=$, $<$, or $\leq$.

- Static Formulas: $SF ::= AF \mid SF \wedge SF \mid \neg SF$
- Temporal Formulas: $TF ::= SF \mid TF \wedge TF \mid \neg TF \mid \mathbf{AX}TF \mid \{\mathbf{A} \mid \mathbf{E}\}[TF\ \mathbf{U}\ TF]$

A model is given by:

- States: $S = 2^{Ap}$.
- Transition relation: $R \subseteq S \times S$
- Initial states: $S_0 \subseteq S$
- Valuation mapping for atomic propositions $V : Ap \times S \rightarrow \{0, 1\}$

The semantics for the logic is given by:

- Propositional formula interpretation: $P : Prop \times S \rightarrow \{0, 1\}$

  $$P(p_i, s) = V(p_i, s); \quad P(f_1 \wedge f_2) = P(f_1, s) \wedge P(f_2, s); \quad P(\neg f, s) = \neg P(f, s)$$

- Word interpretation: $W : Word \times S \rightarrow N$

  $$W((f_0, f_1, \ldots, f_n), s) = \sum_{i=0}^{n} P(f_i, s) 2^i$$

- Expression interpretation: $E : Exp \times S \times S \rightarrow N$. In the following, s' is needed because it is possible to have the next state value of a word in an expression.

  $$E(e_1 \odot e_2, s, s') = E(e_1, s, s') \odot E(e_2, s, s')$$
  $$E(\text{if } f \text{ then } e_1 \text{ else } e_2, s, s') = \text{if } (s \models f) \text{ then } E(e_1, s, s') \text{ else } E(e_2, s, s')$$
  $$E(w, s, s') = W(w, s)$$
  $$E(\text{next}(w), s, s') = W(w, s')$$

- Atomic formula interpretation. Because of the nondeterministic behavior of the system, there can be more than one possible next state for a given state. Therefore, a path quantifier is needed in order to quantify over the next state that appears in the semantics of the expressions.

  $$s \models p_i \Leftrightarrow V(p_i, s) = 1$$
  $$s \models \mathbf{A}(e_1 \sim e_2) \Leftrightarrow \forall s'.R(s, s') \rightarrow E(e_1, s, s') \sim E(e_2, s, s')$$
  $$s \models \mathbf{E}(e_1 \sim e_2) \Leftrightarrow \exists s'.R(s, s') \wedge E(e_1, s, s') \sim E(e_2, s, s')$$

- The semantics of SF and TF are the same as in CTL.

This logic can naturally be divided into three layers. The top layer contains atomic formulas, static formulas and temporal formulas. The second layer contains words and expressions. The third layer contains atomic propositions and propositional formulas. All of the objects in the top and bottom layers are boolean functions while the objects in the second layer are functions that map boolean vectors into the integers. Therefore, in the word level model checking system, all of atomic propositions, propositions, atomic formulas, static formulas and temporal formulas are represented as BDDs; while words and expressions are represented as hybrid decision diagrams.

4. Arithmetic operations on hybrid decision diagrams

In order to be able to perform model checking on the logic discussed in the previous section, it is desirable to implement various operations on hybrid decision diagrams. We consider scalar multiplication, addition and multiplication of two functions, and the if-then-else operation. Although we only discuss a special kind of hybrid decision diagrams in this and the following section, similar algorithms exist for handling general hybrid decision diagrams as well. As discussed in the previous section, we use a uniform hybrid transformation for all functions. Let the transformation matrix be H.

We use f' to denote the result after applying the hybrid transformation to a function f. Scalar multiplication is simple to perform.

$$(c \cdot f)' = H \times (c \cdot f) = c \cdot (H \times f) = c \cdot f'$$

Finding the sum of two function is also simple.

$$(f + g)' = H \times (f + g) = H \times f + H \times g = f' + g'$$

Next, we consider how to perform multiplication. Let the top level variable is x_i. Suppose

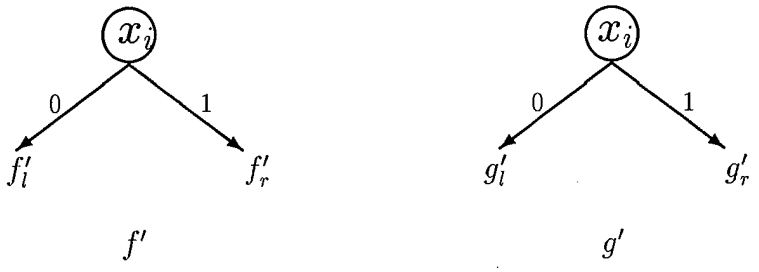


Figure 1: BDDs for f' and g'

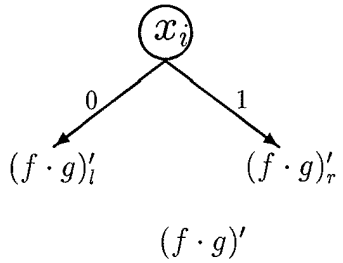


Figure 2: BDD of $(f \cdot g)'$

f', g' are shown in Figure 1, and the resulting function $(f \cdot g)'$ is shown in Figure 2. There are two possibilities. If x_i is a control signal, the identity transformation is used at this level.

Then

$$(f \cdot g)'_l = (f \cdot g)'|_{x_i=0} = (f|_{x_i=0} \cdot g|_{x_i=0})' = (f_l \cdot g_l)'$$
$$(f \cdot g)'_r = (f \cdot g)'|_{x_i=1} = (f|_{x_i=1} \cdot g|_{x_i=1})' = (f_r \cdot g_r)'$$

When x_i is a data bit, the inverse Reed-Muller transformation is used at this level. In this case, the computation is more complicated.

$$\begin{aligned}
(f \cdot g)'_l &= (f \cdot g)'|_{x_i=0} = (f|_{x_i=0} \cdot g|_{x_i=0})' = (f_l \cdot g_l)' \\
(f \cdot g)'_r &= (f \cdot g)'|_{x_i=1} - (f \cdot g)'|_{x_i=0} \\
&= (f|_{x_i=1} \cdot g|_{x_i=1})' - (f|_{x_i=0} \cdot g|_{x_i=0})' \\
&= ((f_l + f_r) \cdot (g_l + g_r))' - (f_l \cdot g_l)' \\
&= (f_r \cdot g_l)' + (f_l \cdot g_r)' + (f_r \cdot g_r)'
\end{aligned}$$

Since both $(f \cdot g)'_l$ and $(f \cdot g)'_r$ can be computed in term of $(f_l \cdot g_l)', (f_l \cdot g_r)', (f_r \cdot g_l)'$, and $(f_r \cdot g_r)'$, we can compute the transformation of the product in a recursive manner. If we store these intermediate results, the total number of recursive calls to compute $(f \cdot g)'$ will be at most $|f'||g'|$. Because of the additions that are needed in the computation, the worst case complexity can still be exponential. However, in practice, this algorithm works quite well.

Likewise, the recursive computation of the if-then-else operation can be given as follows. If the top variable x_i is a control signal,

$$(\text{if } c \text{ then } f \text{ else } g)'_l = (\text{if } c|_{x_i=0} \text{ then } f_l \text{ else } g_l)'$$

$$(\text{if } c \text{ then } f \text{ else } g)'_r = (\text{if } c|_{x_i=1} \text{ then } f_r \text{ else } g_r)'$$

When x_i is data bit,

$$(\text{if } c \text{ then } f \text{ else } g)'_l = (\text{if } c|_{x_i=0} \text{ then } f_l \text{ else } g_l)'$$

$$\begin{aligned}
&(\text{if } c \text{ then } f \text{ else } g)'_r = \\
&(\text{if } c|_{x_i=1} \text{ then } f_l \text{ else } g_l)' + (\text{if } c|_{x_i=1} \text{ then } f_r \text{ else } g_r)' - (\text{if } c|_{x_i=0} \text{ then } f_l \text{ else } g_l)'
\end{aligned}$$

5. Equations and inequalities

Model checking for word level properties also requires computing the set of assignments that satisfy $f_1 \sim f_2$, where $\sim$ can be one of $=, \neq, <, \leq, >$, or $\geq$. Finding the set of assignments that satisfy an inequality can be reduced to the problem of finding the set of assignments that make a function f positive. Equations can be handled in a similar manner. A straightforward way of solving the problem is to convert f to an MTBDD and then pick the terminal nodes with the correct sign. However, this does not work very well in general, because some functions have MTBDDs with exponential size but hybrid BDDs of polynomial size. For example, let $f_1 = \sum_{i=0}^{m} x_i 2^i$ and $f_2 = \sum_{j=0}^{m} y_j 2^j$. Both of these functions and their difference have linear size BMDs. The BDD for the set of assignments satisfying $f_1 - f_2 > 0$ also has linear size. But the MTBDD size for $f_1 - f_2$ is exponential.

We have developed an algorithm that can substantially reduce the cost for computing arithmetic relations between certain functions. Suppose that we want to compute the set of assignments that satisfies $f > 0$. Each branch in the hybrid decision diagram for f corresponds to a subset of variable assignments. If the maximum value of a branch is less than or equal to 0, then none of the assignments in this branch satisfy the inequality. If the minimum value of a branch is greater than 0, then all assignments in this branch satisfy the inequality. In both cases, we avoid checking the signs of the individual assignments in the branch.

To obtain a good algorithm for this problem, it is important to be able to compute upper and lower bounds for a branch in an HDD. An algorithm for this purpose is given below. If the intermediate results are stored, the algorithm takes time linear in the number of HDD nodes.

```
bound_values(f, upper, lower)
begin
  if(f is terminal node)
    upper = lower = f.value;

  if(Top level is BMD)
    lower = min(lower(left(f)), lower(left(f)) + lower(right(f)));
    upper = max(upper(left(f)), upper(left(f)) + upper(right(f)));
  else
    lower = min(lower(left(f)), lower(right(f)));
    upper = max(upper(left(f)), upper(right(f)));
end
```

The improved algorithm for computing the BDD for the set of assignments that make the function f positive is given below. A similar algorithm is used to find the set of assignments that make a function zero.

```
bdd greater_than_0(f)
begin
  if(f is terminal node)
    if(f.value > 0) return(True);
    else return(False);

  bound_values(f, upper, lower);
  if(upper <= 0) return(False);
  if(lower > 0)  return(True);

  left = greater_than_0(left(f));
  if(top level is BMD)
    right = greater_than_0(left(f) + right(f));
  else
```

```
    right = greater_than_0(right(f));
  return(hdd_if_then_else(level(f), left, right));
end
```

The improved algorithm works extremely well for verification of arithmetic circuits. The following theorem guarantees the efficiency of this algorithm for the set of *linear expressions.* Most of the formulas that occur during the verification of the SRT division algorithm are in this class. These expressions have the form $f = \sum_{i=1}^{m} c_i f_i$, where $f_i = \sum_{j=0}^{n} x_{ij} 2^j$ for $1 < i < m$ and the c_i's are integer constants. Suppose all variables are data variables, then the Hybrid Decision Diagrams are identical to BMDs. We use the variable ordering $x_{1n}, x_{2n}, \ldots, x_{mn}, \ldots, x_{10}, x_{20}, \ldots, x_{m0}$. Because $f|_{x_{ij}=1} - f|_{x_{ij}=0} = c_i 2^j$ is a constant, the HDD for f is shown in Figure 3.

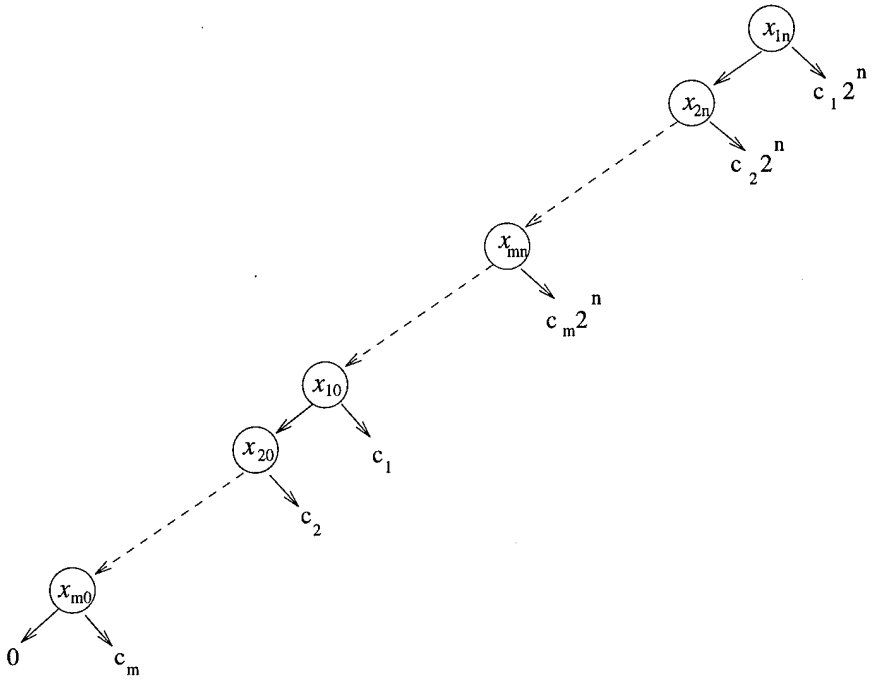


Figure 3: BMD for $\sum_{i=1}^{m} c_i f_i$

Lemma 1 *The number of recursive calls to the* `greater_than_0` *procedure for computing the BDD for f at each level cannot exceed* $4(\sum_{i=1}^{m} |c_i|)$.

Proof: Suppose we consider the recursive calls to the BMD nodes that has x_{ij} as the top variable. The inverse transformation matrix for BMD nodes is the 2×2 Reed-Muller matrix $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. Thus, the recursive calls in the procedure `greater_than_0` apply to either the left child or the sum of both children. The BMD nodes that are recursively called with x_{ij} as top variable must be the sum of the sub-BMD in Figure 3 with top variable x_{ij} and some of the right children of ancestors of the sub-BMD. The right children of all of the ancestor nodes of this sub-BMD are constant nodes with the value $c_k 2^l$ where $1 \leq k \leq m$ and $l \geq j$. The

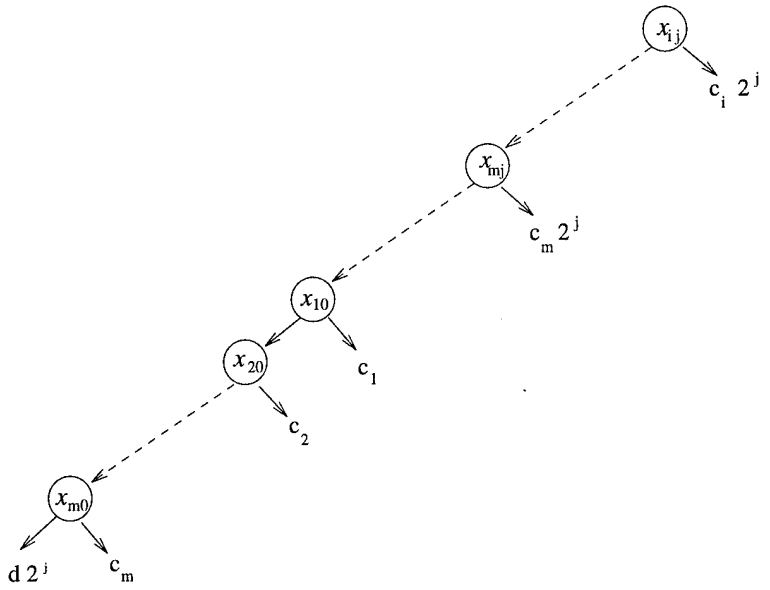


Figure 4: BMD nodes explored at level x_{ij}

sum of those right children can be rewritten in the form $d2^j$ where d is an integer constant. Therefore the BMD nodes with top variable x_{ij} have the form shown in Figure 4.

$$\text{Let } c'_k = \begin{cases} c_k & c_k \geq 0 \\ 0 & \text{otherwise} \end{cases} \text{ and } c''_k = \begin{cases} 0 & c_k \geq 0 \\ c_k & \text{otherwise} \end{cases}$$

When we apply the procedure `bound_values` to this BMD, the upper bound computed is equal to $d2^j + \sum_{l=0}^{j-1} \sum_{k=1}^{m} c'_k 2^l + \sum_{k=i}^{m} c'_k 2^j$. This can be proved by induction on the structure of the BMD. The base case is trivial. For the induction step, consider the node with the variable x_{ij}. There are two cases. The first case is when $i < m$. In this case, by induction hypothesis, `upper(left(f))` is equal to $d2^j + \sum_{l=0}^{j-1} \sum_{k=1}^{m} c'_k 2^l + \sum_{k=i+1}^{m} c'_k 2^j$. Since the right branch is a constant, `upper(right(f))` is $c_i 2^j$. Therefore,

$$\begin{aligned}
\texttt{upper} &= \max(\texttt{upper(left(f))}, \texttt{upper(left(f))} + \texttt{upper(right(f))}) \\
&= \texttt{upper(left(f))} + \text{if } \texttt{upper(right(f))} >= 0 \text{ then } \texttt{upper(right(f))} \text{ else } 0 \\
&= d2^j + \sum_{l=0}^{j-1} \sum_{k=1}^{m} c'_k 2^l + \sum_{k=i+1}^{m} c'_k 2^j + (\text{if } c_i >= 0 \text{ then } c_i \text{ else } 0)2^j \\
&= d2^j + \sum_{l=0}^{j-1} \sum_{k=1}^{m} c'_k 2^l + \sum_{k=i+1}^{m} c'_k 2^j + c'_i 2^j \\
&= d2^j + \sum_{l=0}^{j-1} \sum_{k=1}^{m} c'_k 2^l + \sum_{k=i}^{m} c'_k 2^j
\end{aligned}$$

Similar proof can be obtained for the other case when $i = m$. In the same way, we are able to prove that the lower bound computed by the procedure is $d2^j + \sum_{l=0}^{j-1} \sum_{k=1}^{m} c''_k 2^l + \sum_{k=i}^{m} c''_k 2^j$. Hence

$$\texttt{upper} = d2^j + \sum_{l=0}^{j-1} \sum_{k=1}^{m} c'_k 2^l + \sum_{k=i}^{m} c'_k 2^j$$

$$
\begin{aligned}
&\leq d2^j + \sum_{l=0}^{j}\sum_{k=1}^{m} c'_k 2^l \\
&= d2^j + \sum_{k=1}^{m} c'_k (2^{j+1} - 1) \\
&\leq d2^j + \sum_{k=1}^{m} c'_k 2^{j+1} \\
&= 2^j (d + 2\sum_{k=1}^{m} c'_k)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{lower} &= d2^j + \sum_{l=0}^{j-1}\sum_{k=1}^{m} c'''_k 2^l + \sum_{k=i}^{m} c''_k 2^j \\
&\geq d2^j + \sum_{l=0}^{j}\sum_{k=1}^{m} c''_k 2^l \\
&= d2^j + \sum_{k=1}^{m} c''_k (2^{j+1} - 1) \\
&\geq d2^j + \sum_{k=1}^{m} c''_k 2^{j+1} \\
&= 2^j (d + 2\sum_{k=1}^{m} c''_k)
\end{aligned}
$$

If $d \leq -2\sum_{k=1}^{m} c'_k$, then `upper` is negative or 0 and the algorithm will return constant false. Likewise, if $d > -2\sum_{k=1}^{m} c''_k$, `lower` is positive and the algorithm will return constant true. Therefore, the recursive calls to the children can only occur when $-2\sum_{k=1}^{m} c'_k < d \leq -2\sum_{k=1}^{m} c''_k$. Since d is integer, there can be at most $2 \times (-2\sum_{k=1}^{m} c''_k + 2\sum_{k=1}^{m} c'_k) = 4\sum_{k=1}^{m} |c_k|$ recursive calls. □

Theorem 2 *The complexity of* `greater_than_0` *for* f *is* $O(n^2 \sum_{k=1}^{m} |c_k|)$.

Proof: There are n levels. Each level takes $4\sum_{k=1}^{m} |c_k|$ recursive calls. Each recursive call takes time $O(n)$ to compute the upper and lower bound values. Therefore, the total time is $O(n^2 \sum_{k=1}^{m} |c_k|)$. □

In the case of linear inequalities, all the new BMDs that are generated have the form of $c + g$, where c is a constant and g is an existing BMD. If we remember the constant without actually adding it to the BMDs, we are able to avoid generating new BMD nodes. After introducing this technique, the complexity for compute `greater_than_0(f)` can be further reduced to $O(n \sum_{k=1}^{m} |c_k|)$.

6. Model Checking for Word Level Properties

Model checking is a technique of finding the set of states in a state-transition graph where a given CTL formula is true. There is a program called EMC that solves this problem using

efficient graph-traversal techniques. If the model is represented as a state-transition graph, the complexity of the algorithm is linear in the size of the graph and in the length of the formula. The algorithm is quite fast in practice [6, 7]. However, an explosion in the size of the model may occur when the state-transition graph is extracted from a finite state concurrent system that has many processes or components. In *symbolic model checking systems* [5], BDDs are used to represent the transition relations and sets of states. The model checking process is performed by fixpoint operations on these BDDs. By using symbolic model checking techniques, the size of the transition systems that can be verified has increased dramatically. Although such techniques have been successful in verifying control logic, they cannot be directly used for verifying arithmetic circuits. This is because expressions that involve words with integer values cannot be handled properly.

Now that we are able to handle arithmetic operations and arithmetic relations, it is possible to extend the symbolic model checking algorithm so that it can handle word level properties. BDDs for the transition relation and all propositions are generated in exactly the same way as in the original symbolic model checking system. The hybrid decision diagram representation of a word $(f_0, f_1, \ldots, f_n)$ can be computed as

$$\sum_{i=1}^{n} (\text{if } f_i \text{ then } 2^i \text{ else } 0)$$

using the operations mentioned above. Although this process is exponential in the worst case, it works fairly well in practice. The hybrid decision diagram representation of most expressions can be computed using the techniques discussed above. The only exception is the *next* operation, which can be performed by variable substitution. The substitution replaces all of the current state variables in the hybrid decision diagram for the word by their corresponding next state variables. The algorithm to obtain the BDD representing the set of variable assignments that make an algebraic relation true can be used to compute the BDD for atomic formulas. After the BDD representation for the atomic formulas is generated, the BDDs for static formulas and temporal formulas are computed in the same way as in ordinary model checking. In particular, the fixpoint computations are exactly the same in both cases.

Since we have used the same algorithm to compute the transition relation as in the ordinary model checking algorithm. The word level model checking algorithm does not work well when the transition relation does not have a concise representation. As an example, let's consider a multiplier. Let x and y be the input registers and z be the output register. Suppose the transition relation can be represented as follows:

$$Tr(x, y, z) = Tr'(x, y) \wedge (\text{next}(z) = x \times y)$$

Obviously, the BDD representation of the transition relation has exponential size since the BDD representation of the middle bit of a multiplier is exponential. This problem can sometimes be avoided by conjunctive decomposition of the transition relation. Let $\bar{x}, \bar{y}$, and $\bar{z}$ be the state variables that encode the current state value of x, y and z, respectively. Let $\bar{x}', \bar{y}'$, and $\bar{z}'$ be the state variables that encode the next state value of x, y and z. Suppose that we want to verify a word level property of the form $f(x, y, z)$. There may be appearances of next(z); if so, we can replace them by $x \times y$ at the word level and obtain a new formula.

Hopefully, the resulting formula will be independent of z and the BDD representation of the formula can be denoted as $f'(\bar{x}, \bar{y})$. In this case, we can use Tr' as the transition relation to perform the fixpoint operations. Even if f' depends on some bits of z, we can often obtain a much simpler transition relation by eliminating the conjuncts that give the values of bits that are not needed.

7. Verification of an SRT radix 4 division circuit

By using the word level model checking system, we have successfully verified circuits for division and square root computation that are based on the SRT algorithm used by the Pentium. We are able to handle both the control logic and the data paths. The division circuit that we investigated has 5 states, *idle, init, loop, last* and *rem*. The state transition graph for these states are shown in Figure 5. This circuit can perform two different operations

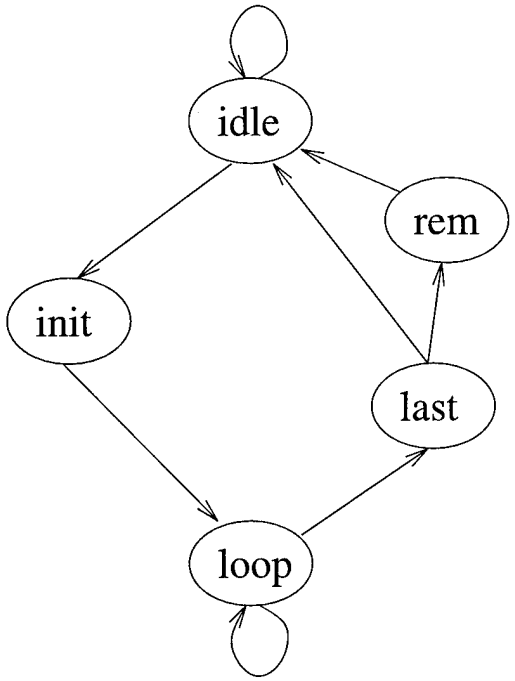


Figure 5: The controlling states for the division circuit

division and *remainder*. When the operation is *division*, the steps in the computation are

$$idle \rightarrow init \rightarrow loop^* \rightarrow last \rightarrow idle$$

When the operation is *remainder*, the steps are

$$idle \rightarrow init \rightarrow loop^* \rightarrow last \rightarrow rem \rightarrow idle$$

Figure 6 gives the data path of the circuit at loop state. All the words have 70 bits. However, only leading bits of the partial remainder and multiples of divisor are used to compute the quotient digit for the next cycle.

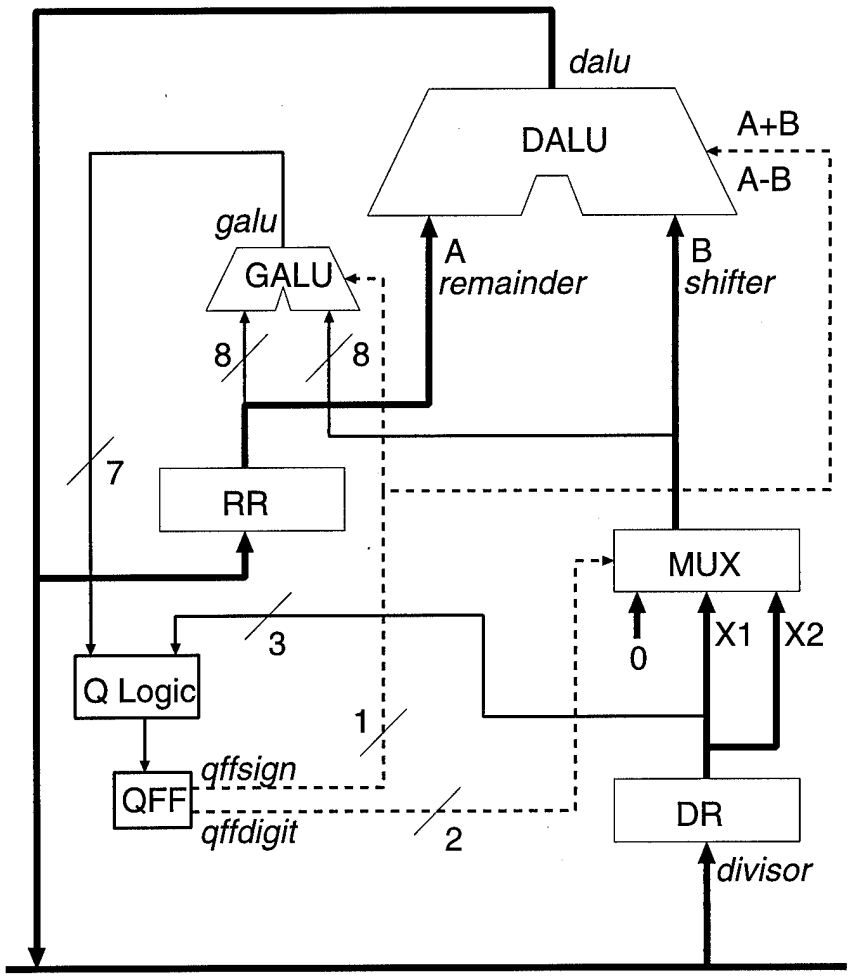


Figure 6: The data path for the division circuit at loop state

We have verified the circuit with both control logic and the data path. All states of the finite state machine have been checked. Let r be partial remainder, q be quotient, d be the divisor. We have checked the properties

- The expression $r + q \cdot d$ always equals the dividend.
- The computation does not overflow. This is guaranteed by $-\frac{8}{3}d \leq r \leq \frac{8}{3}d$.

For example, we have proved that at init state, the remainder is the dividend and the quotient is zero. Therefore, the initial value for $r + q \cdot d$ equals the dividend. Moreover, the inequality mentioned above holds at the init state.

```
SPEC AG(state = init -> r = dividend & q = 0)

SPEC AG(state = init -> (-8) * d <= 3 * r <= 8 * d)
```

```
    g1  (remainder -- first 7 bits)
    g2
    g3  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1
        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
    g4  0  0  1  1  0  0  1  1  0  0  1  1  0  0  1  1  0  0  1  1  0  0  1  1
    g5  1  1  1  1  0  0  0  0  1  1  1  1  0  0  0  0  1  1  1  1  0  0  0  0
    g6  0  0  0  0  1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  1  1  1  1
    g7  1  1  1  1  1  1  1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0

1.000  -- -- -- -- -- -2 -2 -2  A -1 -1  0  0  1  1  2  2  2 -- -- -- -- -- --
1.001  -- -- -- -- -- -2 -2 -2  B -1 -1  0  0  1  1  C  2  2  2 -- -- -- -- --
1.010  -- -- -- -- -2 -2 -2 -2 -1 -1  D  0  0  1  1  1  2  2  2  2 -- -- -- --
1.011  -- -- -- -2 -2 -2 -2  B -1 -1  D  0  0  1  1  1  2  2  2  2 -- -- -- --
1.100  -- -- -- -2 -2 -2 -2 -1 -1 -1  0  0  0  E  1  1  C  2  2  2  2 -- -- --
1.101  -- -- -2 -2 -2 -2 -2 -1 -1 -1  0  0  0  0  1  1  1  2  2  2  2  2 -- --
1.110  -- -2 -2 -2 -2 -2  B -1 -1 -1  0  0  0  0  1  1  1  2  2  2  2  2 -- --
1.111  -- -2 -2 -2 -2 -2 -1 -1 -1 -1  0  0  0  0  1  1  1  1  2  2  2  2  2 --

(divisor -- first 4 bits)              A = -(2 - g2 * g1)
                                       B = -(2 - g2)
                                       C = 1 + g2
                                       D = -(1 - g2)
                                       E = g2
```

Table 1: The quotient prediction table for the division circuit

We have also proved that the inequality always holds in the loop states, and that $r + q \cdot d$ is invariant with respect to left shifting.

```
SPEC AG(state = loop -> A[((-8) * d <= 3 * r <= 8 * d) U state = last])
```

```
SPEC AG((state = loop & ((-8) * d <= 3 * r <= 8 * d))
        -> A((r + q * r) * 4 = next(r + q * r)))
```

The above properties are sufficient to guarantee that in the loop state, $r + q \cdot d$ always equals the dividend after left shifting. Similar properties are proved for the *last* and *rem* states. In addition, we have verified a circuit for computing square roots. The total number of state variables for the circuit that we verify exceeds 600 (which is much larger than any circuit previously checked by SMV).

8. Directions of Future Research

We have verified a floating point division circuit based on the SRT algorithm using the word level model checker. We plan to experiment on more circuits. Possible applications include the floating point multiplier, floating addition, etc.

Our algorithm for solving arithmetic relations works extremely well for linear equations and inequalities. Although the current algorithm can handle some nonlinear equations and inequalities as well, it may be possible to extend this algorithm or to find a new algorithm that can handle more complicated nonlinear equations and inequalities.

There is still one problem with this technique. It can only be used for circuits that maintain the exact value of the data. When rounding occurs, the functions become less regular and the size of hybrid BDD representation is likely to explode. In these cases, the new value obtained after rounding can be described by a system of inequalities, and the verification process reduces to solving such systems. In another research project, we have built a theorem prover based on symbolic computation system Mathematica. The theorem prover is called *Analytica* [9] and is quite good at handling equations and inequalities. We believe that after some modification, Analytica will be useful for solving the inequalities that arise because of rounding in computer arithmetic.

References

[1] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of the 1993 Proceedings of the IEEE International Conference on Computer Aided Design.* IEEE Computer Society Press, November 1993.

[2] R. Bellman. *Introcution to matrix analysis*, chapter 5. McGraw-Hill, 1970.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[4] R. E. Bryant and Y. A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference.* IEEE Computer Society Press, June 1995.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[6] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[8] E. M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Proceedings of the 30th ACM/IEEE Design Automation Conference.* IEEE Computer Society Press, June 1993.

[9] E. M. Clarke and X. Zhao. Analytica: A theorem prover for mathematica. *The Journal of Mathematica*, 3(1), 1993.

[10] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of switching functions based on ordered kroenecker functional decision diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, June 1994.

[11] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. To appear.

[12] D. E. Muller. Application of boolean algebra to switching circuit design and error detection. *IRE Trans.*, 1:6–12, 1954.